An Efficient Algorithm for Link Distance Problems (Extended Abstract)

Yan Ke
Department of Computer Science
The Johns Hopkins University
Baltimore, MD 21218

Abstract

The *link distance* between two points inside a simple polygon P is defined to be the minimum number of edges required to form a polygonal path inside P that connects the points. A *link furthest neighbor* of a point $p \in P$ is a point of P whose link distance is the maximum from p. The *link center* of P is the collection of points whose link distances to their link furthest neighbors are minimized. We present an $O(n \log n)$ time and $O(n)$ space algorithm for computing the link center of a simple polygon P, where n is the number of vertices of P. This improves the previous $O(n^2)$ time and space algorithm. Our algorithm essentially sweeps a *chord* through the polygon and spends $O(\log n)$ time at each step. We demonstrate that the output of the algorithm, a sequence of sets of chords, is a powerful tool for solving several other link distance problems.

1 Motivations and Definitions

The study of link distance problems was motivated by the following robot motion-planning problem. Consider a point-size robot who wants to move between two points within a polygon P. Suppose the robot can perform two types of movements: a straight line motion and a pure rotation. Therefore a feasible path for the robot is a polygonal chain that connects the two points in P. Further suppose that straight line motion is cheap, while a pure rotation is computationally expensive. Therefore a "good" path for the robot is the one that has as few number of turns, or number of edges (links), as possible.

We now give several formal definitions concerning the link distance between points. *A minimum link path* between two points p_1 and p_2 is a polygonal path that connects p_1 and p_2 in P with the minimum number of edges; this number is *the link distance* between p_1 and p_2. The problem of computing a minimum link path was solved by Suri [12], in which an $O(n \log \log n)$ time algorithm, or $O(n)$ time if a triangulation of P is precomputed [11][14], was presented, where n is the total number of vertices of P. Given a point $p \in P$, its *link furthest neighbor* is a point of P whose link distance from p is the maximum over all other points of P. The *link center* of P is the collection of points whose link distances to their link furthest neighbors are minimum over all other points of P. The *link radius* of P is the link distance from a point in the link center to its link furthest neighbor. The *link diameter* of P is the maximum link distance between two points of P, or equivalently, the maximum link distance from a point to its link furthest neighbor over all points of P.

We now list some of our results and compare them with the best previous ones.

- Computing the link center and radius in $O(n \log n)$ time and $O(n)$ space (best previous result: $O(n^2)$ time and $O(n^2)$ space [10] [1]);
- Computing the link diameter in $O(n \log n)$ time and $O(n)$ space (same complexities [13]);
- Solving the *all link furthest neighbor problem* in $O(n \log n)$ time, that is, report a link furthest neighbor for each vertex of P (best previous result: $O(n^2)$ time and $O(n)$ space [13]);
- Preprocessing P in $O(n \log n)$ time and $O(n)$ space such that given a query point in P, its link furthest neighbor and the link distance between

[1] As this abstract was been written, we learned of [2], which presents an $O(n \log n)$ time and $O(n)$ space algorithm to compute a non-empty region within the link center. Later they claimed that the algorithm was improved to find the entire link center.

them can be found in $O(\log n)$ time (best previous result: $O(n \log\log n)$ time and $O(n)$ space preprocessing, and $O(n)$ query time [13]);

Besides these, our algorithm can also solve some extensions of the link distance problems. First we need more definitions. A minimum link path between a point p and a line segment s is a polygonal path that connects p to a point on s within P with the minimum number of edges; this number is the link distance between p and s. A link furthest neighbor of a line segment $s \in P$ is a point of P whose link disance from s is the maximum over all other points of P. A *link central segment* of P is a line segment s whose link distance to its link furthest neighbors is the minimum over all other segments in P;

In the following, we list our results for two problems that, as far as we know, have not been studied previously.

- Computing a shortest link central segment in $O(n \log n)$ time and $O(n)$ space;
- Preprocessing P in $O(n \log n)$ time and $O(n)$ space such that given a query segment, its link furthest neighbor and the link distance between them can be found in $O(\log n)$ time;

All these results are achieved essentially by one algorithm, which uses a *chord* to sweep through the polygon P in a recursive fashion. It generates a sequence of sets of chords that satisfy a certain *Intersection Properties*, Property I and I^*, which provides all the information we need to solve the mentioned problems.

In section 2, we introduce the concept of Property I and I^*, and show how to solve all the link distance problems by using the sets of chords with this property. Then in Section 3 and 4, two algorithms are described that generate the desired sets of chords: a brute-force algorithm and a simple $O(n^2 \log n)$ *sweep-chord* algorithm. Section 5 improves the time complexity to $O(n \log n)$. Finally in Section 6, we show our results for the additional extensions of link distance problems.

2 Intersection Property

We start with some basic definitions. A *chord* of P is a line segment in the interior of P except for its two endpoints, which then must be on the boundary of P. Therefore a chord c divides P into exactly two subpolygons, and one of them (specified later) is called the *component* of c, denoted by $P(c)$.

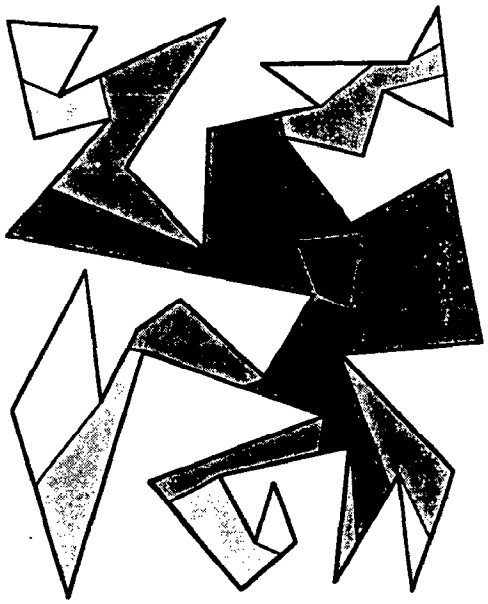

Figure 1: The partition of P

A sequence of sets of chords $C_1, C_2, ..., C_k$ is said to have the *Intersection Property*, or Property I, if the following statment holds:

> *For any point $p \in P$, p has at most link distance i, $1 \le i \le k$, to every point of P iff it is contained in the intersection region $R_i = \bigcap_{c \in C_i} P(c)$.*

Suppose such C_i is available, $1 \le i \le k$. Let us see how can we use them to solve the previously mentioned link distance problems. First, since any point $p \in P$ can reach every point of P by at most d links, where d is the link diameter of P, we immediately conclude that $d = k + 1$. Clearly the link center is R_r, where r is the link radius and satisfies the inequality $\lceil d/2 \rceil \le r \le \lceil d/2 \rceil + 1$ [10]. Therefore the link center and radius can be computed in $O(N \log N)$ time, where N is the total number of chords in the sets. (It can be shown that $N = \Omega(n)$.) Next define $R'_r = R_r$, $R'_i = R_i - R_{i-1}$ for $r + 1 \le i \le d - 1$, and $R'_d = P - R_{d-1}$. Then any points in R'_i can reach all points of P by i links, but not $i - 1$ links. In other words, they have link furthest neighbors that are i links away. These regions R'_r, R'_{r+1}, ..., R'_{d-1}, R'_d form a partition of P. See Figure 1 for an example: the black region is the collection of points having link distance 4 to their link furthest neighbors; this region is the link center and 4 is the link radius of

the polygon. The dark shaded region is the collection of points having link distance 5 to their link furthest neighbors. The shaded region is the collection of points having distance 6 to their furthest neighbors. The light shaded region having distance 7 to their neighbors. The blank region having distance 8 to their neighbors, 8 is the link diameter of the polygon.

Using the point-location techniques [3], we can preprocess this partition in $O(N \log N)$ time and $O(N)$ space such that given a query point p, its link furthest neighbor and the link distance between them can be obtained in $O(\log N)$ time.

A stronger version of the Intersection Property, Property I^*, is defined as follows:

> *For any line segment $s \in P$, s has at most link distance i, $1 \leq i \leq d-1$, to every point of P iff it intersects the component $P(c)$ for each chord $c \in C_i$.*

We use the following results from [8]: given a chord set C with size k in a n-gon P, we can decide in $O(n \log n + k \log k)$ time and $O(n+k)$ space, or $O(\log n + \log k)$ time if an $O(n \log n + k \log k)$ time and $O(n+k)$ space preprocessing of C and P is allowed, whether there exists a line segment intersecting the component of each chord in C; if the answer is positive, then we can find a shortest one in $O(n \log n + k \log k)$ time and $O(n+k)$ space. Therefore computing a shortest link central segment costs $O(N \log N)$ time once we have the chord set C_i, $1 \leq i \leq d-1$. We can also solve the query problem as follows: first preprocess C_i, for $1 \leq i \leq d-1$, by the algorithm in [8]. Then when a query segment s is given, choose an arbitrary point $p \in s$ and find the link distance, say i, between p and its link furthest neighbor; then the link distance from s to its link furthest neighbor is either i or $i-1$, which can be found out in additional $O(\log N)$ time.

Given the power of sequence of chords satisfying Property I and I^*, there are two goals: to find sets of total size $N = O(n)$, and to do so quickly, in $O(n \log n)$ time.

3 A Brute-force Algorithm

In this section, we briefly describe a brute-force algorithm that generates the desired chord sets C_1, C_2, ..., C_{d-1} with $O(n^2)$ total number of chords. For simplicity, we ignore all degeneracies throughout the paper. It is well known that any point $p \in P$ can reach all points of P by i links iff it can reach all vertices of P by i links. we will use this fact implicitly in many places.

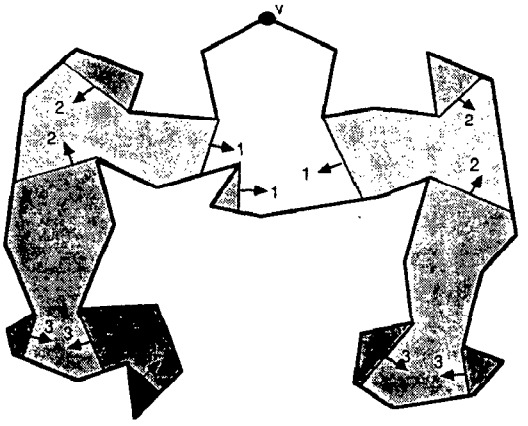

Figure 2: An example of the sets of chords with respect to a vertex v. The arrows on the chords indicate at what sides of their components lie. The number at each chord indicates the index of the set that contains that chord.

We start with one vertex v, and construct its visibility polygon $V(v)$. Let $C_1(v)$ be the set of chords on the boundary of $V(v)$. For each $c \in C_1(v)$, define its component $P(c)$ to be the sub-polygon that contains v. Then $\bigcap_{c \in C_1(v)} P(c) = V(v)$ is exactly the collection of points that can reach v by at most 1 link. We remove $V(v)$ from P and, within its remaining portion, construct the visibility polygon $V(c)$ for each chord $c \in C_1(v)$. Let $C_2(v)$ be the set of newly generated chords with their components defined in the same manner. Then $\bigcap_{c \in C_2(v)} P(c)$ is exactly the collection of points that can reach v by at most 2 links. We can repeat this construction until the entire polygon P is covered. In the end, we have a sequence of chord sets $C_1(v)$, $C_2(v)$, ..., $C_{d(v)-1}(v)$, where $d(v)$ is obviously the link distance from v to its link furthest neighbor. See Figure 2. The entire procedure can be managed to run $O(n)$ time provided that a triangulation of P is given (see [13] for details).

At this time, We need to give a few definitions for our later discussion. We define the *label* of a chord $c \in C_i(v)$ to be i, $1 \leq i \leq d(v) - 1$. A chord c is called a *direct descendant* of another chord c' if c is on the boundary of $V(c')$, i.e., c is generated *from* c'; and c' is called the *direct ancestor* of c. We say that c_k is a *descendant* of c_1 if there is sequence of chords c_k, c_{k-1}, ..., c_2, c_1 such that c_i is a direct descendant of

c_{i-1}, for $i = k, k-1, ..., 3, 2$; and c_1 is an *ancestor* of c_k. There is a simple observation: each chord c has one endpoint incident to a reflex vertex of P, which we call the *origin* of c; the other endpoint we call the *hit point* of c. One can understand this definition by imagining the direct ancestor of c (each chord has a unique direct ancestor) to be a light source; then c is a light ray shooting from its origin to its hit point.

We repeat the above described procedure for each vertex of P. Let $d = \max_{\text{each vertex } v} d(v)$ to obtain the link diameter of P. Then let $C_i = \bigcup_{\text{each vertex } v} C_i(v)$, for $1 \leq i \leq d-1$, to obtain the sets of chords that have Property I and I^*. The total number of chords is clearly $O(n^2)$.

4 A Sweep-chord Algorithm

The improvement over the brute-force algorithm is possible only if we can bound the total number of chords in the sets. Consider a chord $c \in C_i$. We define c to be *redundant* if there exists another chord $c' \in C_i$ such that $P(c') \subset P(c)$. Intuitively, if c is redundant, then $P(c)$ has no contribution to the intersection $\bigcap_{c \in C_i} P(c)$. A chord is *necessary* if it is not redundant. There are immediately two observations. First, the two chords c and c' need not have the same label. c is also redundant when the label of c' is greater than that of c. Second, all descendants of a redundant chord are also redundant.

We now show that there are only $O(n)$ necessary chords in the sets $C_1, ..., C_{d-1}$. Recall that each chord c has one endpoint incident to a reflex vertex v of P. We define c to be a *clockwise* chord if its component lies in clockwise direction about v. Similarly, define c to be *counterclockwise* if its component lies in counterclockwise direction about v. We next show that there are at most two necessary chords incident to v, one is clockwise and the other counterclockwise. Suppose there are two, say clockwise, chords c_i and c_j incident to v. If they have the same label, then one of them is redundant. So wlog, let i and j be their labels respectively such that $i < j$. Then we have two cases. If c_j lies in clockwise direction from c_i, then c_i is redundant. Now consider the case when c_i lies in clockwise direction from c_j. Consider the direct ancestor c'_j of c_j. We know c'_j is contained in $P(c_j)$. Suppose it is not contained in $P(c_i)$. Then c'_j must intersect c_i at a point p that can see a small neighborhood of v, including some points not in $P(c_j)$. This is not possible. Therefore c'_j is contained in $P(c_i)$, implying that $P(c'_j)$ is contained in $P(c_i)$. Since $j > i$, the label of c'_j is no less than i. So c_i is redundant. We thus conclude that there are $O(n)$ necessary chords.

Our algorithm generates $O(n)$ number of chords that include all the necessary ones. We give one definition first. We say a chord c is *downward* if its origin is above its hitpoint (again, imagine the direct ancestor of c to be a light source, then c is a ray shooting from its origin *down* to its hit point); otherwise c is called *upward*.

As a preprocessing step of the algorithm, we first partition P into monotone regions as follows. For each reflex vertex v that either points downwards (we call it a *downward cusp*) or upwards (*upward cusp*), we draw the two horizontal chords that are incident to v. This gives us a monotone partition. Then we define a graph whose nodes correspond to the monotone regions and two nodes are connected by an edge if their corresponding regions are adjacent. We will occasionally use the same notation for a node and its corresponding region. The defined graph is clearly an $O(n)$ size tree. Let T be the obtained embedding of the tree. Within T, if a node t_1 is adjacent to another node t_2 from above, then t_1 is called a *parent* of t_2, and t_2 a *child* of t_1. Two nodes are called *siblings* (*spouses*) if they have a common parent (child). A *top* node is the one that has no parents, and a *bottom* node has no children.

Unlike the brute-force algorithm that generates all the descendants of each vertex, one at a time, the sweep-chord algorithm basically takes care of all downward (or upward) chords incident to each reflex vertex, one at a time, by a recursive sweep of the entire polygon. In this way, the generation of redundant chords is well under control. The general structure of the algorithm is as follows.

ALGORITHM *SWEEP-POLYGON*
1. Let all nodes of T be unmarked;
2. **for** each bottom node t of T **do**
3. *SWEEP-DOWN* (t);
4. **for** each top node t of T **do**
5. *SWEEP-UP* (t);

END of *SWEEP-POLYGON*.

There are two procedures in the algorithm. The procedure *SWEEP-DOWN* (t) basically sweeps the monotone region t downwards, and then marks the node t. The procedure *SWEEP-UP* (t) operates similarly. Since the two procedures are symmetric, we only describe the former one.

PROCEDURE *SWEEP-DOWN* (t)
1. **if** t has two parents t_1 and t_2 **then**

2. **for** $i = 1$ to 2 **do**
3. **if** t_i is not marked by "d" **then**
4. *SWEEP-DOWN* (t_i);
5. **else if** t has one parent t_1 and one sibling t_2 **then**
6. **if** t_1 is not marked by "d" **then**
7. *SWEEP-DOWN* (t_1);
8. **if** t_2 is not marked by "u" **then**
9. *SWEEP-UP* (t_2);
10. Sweep the monotone region t downwards to generate all necessary downward chords within t, then mark t by "d";

END of *SWEEP-DOWN* (t).

A simple observation is that each monotone region is processed downwards and upwards, each exactly once. Within each monotone region, the algorithm sweeps a horizontal chord c_s from top to bottom and stops at each vertex to update the maintained data structure and generate downward chords incident to that vertex. Upon the completion of the algorithm, all necessary chords should be generated. More precisely, Step 10 consists of the following three stages:

- *Initial stage*: Collect the information obtained from t_1 and t_2. Then build up the appropriate data structure;
- *Sweeping stage*: Sweep the monotone region downwards vertex by vertex, and maintain the data structure;
- *Processing stage*: When the entire monotone region has been swept, process the data structure such that they can be used by the sweeping procedure of other monotone regions;

We now sketch the data structure maintained by the algorithm. Let c_s be the current sweeping chord in a downward sweep. To keep track of all the downward chords below c_s, we have to maintain all their ancestors above c_s. More precisely, we call a chord c *active* with respect to c_s if (1) it is above c_s, (2) its component is disjoint from c_s, and (3) it is visible from c_s. Intuitively, these are the chords that have downward direct descendants below c_s to be constructed. It is *sufficient* and *necessary* for the algorithm to maintain all the active chords with respect to c_s to generate the downward chords below c_s. However, we can not afford to maintain all the active chords separately since there are $O(n)$ of them.

The solution is to organize the $O(n)$ number of active chords into two convex chains, which can be more easily manipulated geometrically. First we throw away all the chords whose labels are less than $M-1$, where M is the maximum label of the active chords. Then we obtain a partition of all active chords with label M (the treatment for the chords with label $M-1$ is identical and thus omitted). Within each class of the partition, we take the intersection of the components of all the chords; the boundary of the intersection region in the interior of P is a convex chain. The partition is chosen in such a way that all intersection regions are disjoint. Therefore all convex chains are disjoint and naturally ordered along the boundary of P. It is shown that these convex chains, together with those obtained from active chords with label $M-1$, will give all the necessary downward descendants below c_s. Furthermore, every non- (left or right) extreme convex chain can be replaced by its direct descendants that cross the sweeping chord c_s; we call these direct descendants *critical*. Therefore to generate all necessary downward descendants, the algorithm needs to know the four convex chains (the leftmost and rightmost with label M, and with label $M-1$; these are called the *active chains*, or *ACs* for short), and all critical chords. We called this the *upper data structure* of c_s, or UD for simplicity. The *lower data structure* LD can be similarly defined.

We assume that P has been preprocessed for *bullet shooting* and *geodesic path* queries. The first one is to find the hit point of a query ray from a given a point in a given direction. An $O(n)$ time algorithm, provided that P is triangulated, was given by Chazelle and Guibas that can build an $O(n)$ space data structure with $O(\log n)$ query time [1][5]. The second one is to output the geodesic path between two query points. For this, an $O(n)$ space data structure was proposed by Guibas and Hershberger [4]. It can be constructed in $O(n)$ time, provided that P is triangulated, and performs the query in $O(\log n)$ time, under the condition that the output path will be returned before the next query is made. These two data structures will be extensively used by our algorithm.

As mentioned above, the algorithm performs exactly two sweeps for each node of T, one downward and one upward. Let t be a node of T and R_t be its corresponding monotone region. We use $c_u(t)$ and $c_l(t)$ to denote the upper and lower horizontal boundaries of R_t (each of them can either be a chord, or consists of two chords). We will only describe the downward sweep of t.

1. *Initial stage*: We have three cases:
   a. *t is a top node of T.* We create an empty UD with respect to $c_u(t)$, with ACs assigned the la-

bel 0.

b. *t has two parents t_l and t_r with t_l to the left of t_r.* Then $c_u(t)$ consists of two chords $c_l(t_l)$ and $c_l(t_r)$. By the algorithm, both t_l and t_r have been processed downwards. Therefore both UD_l and UD_r, which are the UDs with respect to $c_l(t_l)$ and $c_l(t_r)$, are available. We obtain the UD with respect to $c_u(t)$ by merging UD_l and UD_r. More precisely, let k_l and $k_l - 1$ be the labels of ACs in UD_l, k_r and $k_r - 1$ the labels of ACs in UD_r. Wlog, let $k_l \leq k_r$. If $k_l = k_r = k$, then the left extreme AC in UD_l and right extreme AC in UD_r, with labels k and $k-1$, form the ACs in UD. If $k_l = k_r - 1$, then the ACs in UD_l with label $k_l - 1$ will be discarded. The left extreme AC with label k_l in UD_l and the right extreme AC with label $k_r - 1$ in UD_r, plus the ACs with label k_r in UD_r, form the ACs in UD. If $k_l \leq k_r - 2$, then ACs in UD_l will completely discarded. the ACs in UD_r will be the ACs in UD. In addition, the set of critical chords are combined correspondingly in each case.

c. *t has one parent t_p and one sibling t_s.* Then $c_l(t_p)$ consists of $c_u(t)$ and $c_u(t_s)$. By the algorithm, t_p (t_s) has been processed downwards (upwards). Therefore both UD_p and LD_s, which are the UD and LD with respect to $c_l(t_p)$ and $c_u(t_s)$ respectively, are available. We obtain the UD with respect to $c_u(t)$ by merging UD_p and UD_s. Here we describe some details. Wlog, let t_s be to the left of t. Notice some critical chords in DL_s may be active with respect to $c_u(t)$. Consider those of them with the maximum label k_s (incidentally, these are upward chords). Let c be the one whose hit point is ordered first clockwise along the boundary of P starting at the left endpoint of $c_u(t_s)$. Let k_p and $k_p - 1$ be the labels of ACs in UD_p. Then we have several cases to consider. If $k_s \leq k_p - 1$, then UD is the same as UD_p. If $k_s = k_p$, then ACs in UD includes the AC's in DU_p with label $k_p - 1$, plus the modified ACs (by c) in DU_p with label $k_p = k_s$. If $k_s = k_p + 1$, then ACs in UD includes c, and the ACs in UD with label k_p. Finally, if $k_s \geq k_p + 2$, then c is the only AC in UD and the entire UD_p is discarded. In addition, the set of critical chords are computed correspondingly in each case.

The important observation is that all ACs discarded in the procedure will produce only redundant chords. Therefore their absence does not effect the correctness of the final output. The critical chord c found in Case c essentially takes care of the situation when a link path travels out the monotone region t_s, makes a turn, and then goes into the monotone region t. We thus call c the T-chord (T stands for turn). It should be mentioned that the T-chord c is not computed in this stage, it was actually found and stored at $c_u(t_s)$ in the processing stage of t_s (which will be described later). Therefore an initial stage of the algorithm can be implemented in $O(\log n)$ time.

2. *Sweeping stage*: Each time the sweeping chord c_s stops at a vertex, we check if any critical chords have become active (that is, above c_s). Then these chords are used to update the current ACs and generate new critical chords. Details can be found in [6].

Within each step of the sweeping stage, it costs $O(n)$ time to find those out-of-date critical chords, and $O(\log n)$ time to update the current ACs. The new ACs must be able to generate all necessary descendants that the old ACs have.

3. *Processing stage*: Again, we have three cases for t:

   a. *t is a bottom node.* Then the processing stage is null.

   b. *t has a spouse t_s and a child t_c.* Then $c_u(t_c)$ consists of $c_l(t)$ and $c_l(t_s)$. Wlog, let t be to the left of t_s. Our task in this case is to find the critical chord c in UD with the maximum label, and whose hit point is ordered last (among all critical chords with the maximum label) clockwise along the boundary of P starting from the left endpoint of $c_l(t)$. The chord c is the T-chord with respect to $c_l(t_s)$.

   c. *t has two children t_l and t_r.* Then $c_l(t)$ consists of $c_u(t_l)$ and $c_u(t_r)$. Our task in this case is to compute the UDs with respect to $c_u(t_l)$ and $c_u(t_r)$ by splitting UD with respect to $c_l(t)$. Notice some critical chords may be active with respect to $c_u(t_l)$, and some be active with respect to $c_u(t_r)$. Using a similar criterion for choosing T-chord in Case c of initial stage, we choose one of them for $c_u(t_l)$, and one for $c_u(t_r)$. These two chords are called S-chords (S stands for straight) The two S-chords will then be used to obtain UDs with respect to $c_u(t_l)$ and $c_u(t_r)$.

A processing stage can be implemented in $O(n \log n)$ time. The algorithm thus takes overall $O(n^2 \log n)$ time. The correctness of the algorithm is based on the fact that at each step, we only discard ACs and critical chords that have no contributions to the necessary chords.

5 The Improvement

The the bottleneck for a more efficient algorithm is the $O(n \log n)$ time complexity for a processing stage and the $O(n)$ time complexity for a step in a sweeping stage. These are the two issues we will deal with in this section.

The algorithm can generate a critical chord in one of the three stages. Suppose at some moment, the algorithm generates a downward critical chord c. We observe that c will not be used by the algorithm until one of the following events occurs:

- In a sweeping stage, c becomes active with respec to the sweeping chord c_s as it moves down;
- In a processing stage, c crosses the sweeping chord c_s that is incident to a downward cusp. At this moment, c may become a T-chord;
- In a processing stage, c crosses the sweeping chord c_s that is incident to an upward cusp. At this moment, c may become a S-chord;

The idea behind the improved algorithm is as follows. We will not main the critical chords in UD (and LD). In stead, we insert a critical chord c, as soon as it is created, into some data structures such that c can be captured later in the algorithm when it is needed. There are two such data structures. The first one is used for the occurrence of the first event. It consists of a collection of convex chains, each for an edge of P. When a critical chord c is created, we first find the edge of P that contains the hit point of c, then update the associated convex chain appropriately. The second data structure consists of two balanced trees T_d and T_u that are used for the second and third events. In the remainder of the section, we will very briefly describe T_u (T_d is similar) and the idea of how it improves our algorithm.

The leaves of T_u are all the upward cusps of the polygon P, and they are arranged according to a linear order defined below after a few definitions. Let v be an upward cusp of P and s be the union of two horizontal chords incident to v. (We call the one to the left the L-chord, and the other one the R-chord; these two chords are also called *paired*). The polygon P is divided by s into three components. The one above s is called the *upper component* of v. For the two remaining components below s, the one to the left of v in its small neighborhood is called the *left component* of v, and the other one the right the *right component* of v. An upward cusp is called a *top* upward cusp if its upper component contains no other upward cusps. Choose an arbitrary top upward cusp v. We define all the upward cusps in its left component to be ordered lower than v, and all those in its right component are ordered higher than v. Then the definition is applied recursively to a top upward cusp in each of its left and right components. All L-chords (R-chords) can also be ordered by their incident upward cusps. Therefore we can also think of leaves of T_u as all L-chords (R-chords). Each internal node of T_u is associated with the following four fields: LT, LS, RT and RS (they stand for left T-chord, left S-chord, right T-chord and right S-chord respectively).

In a moment, we will show that this order can be computed efficiently in $O(n \log n)$ time. First let us see how T_u is used in the improved the algorithm. Suppose a new critical chord c is created at a step of the algorithm. Wlog, let c be a downward critical chord with its component at the left side (for this, c is called the *left* critical chord). Then c is a potential S-chord of some R-chords if c intersects their paired L-chords. So consider all the L-chords that are intersected by c. Let c_a and c_b be the highest and lowest among them along c. In fact, they are the highest and lowest ordered ones among such L-chords. For this reason, they are called the *limiting* chords of c. We store c into the LS fields of a subset of nodes in T_u whose leaves exactly cover all such L-chords (this set of nodes has size $O(\log n)$ and can be found in $O(\log n)$ time). In a similar way, c will also be inserted into the fields LT of an appropriate subset of nodes in T_d. In general, we have the following rules:

1. c is *downward left*: insert into LS fields of T_u and LT fields of T_d;
2. c is *upward left*: insert into LT fields of T_u and LS fields of T_d;
3. c is *downward right*: insert into RS fields of T_u and RT fields of T_d;
4. c is *upward left*: insert into RT fields of T_u and RS fields of T_d;

By the time we want to compute, e.g., a S-chord for a R-chord, we simply locate its paired leaf in T_u and walk up to the root to examine the chords stored in the LS fields of its ancestors. Unfortunately, there could be total $O(n)$ chords stored on the path by the naive approach. However, we show next that at most one chord needs to be stored at each node of T_u.

Suppose a downward left critical chord c_2 is being inserted into the LS field of a node t in T_u when we discover that another chord c_1 is already there. Then

we first compare their labels and discard the one with less label. When there is a tie, we further check the positions of their components. Suppose they are disjoint. Consider their higher limiting chords and the corresponding upward cusps. We discard one of c_1 and c_2 that is contained in the upper component of the upward cusp of the other (it can be shown that such a chord always exists). Now suppose the components of c_1 and c_2 are not disjoint. Then we discard the one whose origin is outside the component of the other.

Wlog, let c_2 be discarded. We have to show that c_2 can not be the *necessary* S-chord for a R-chord c_r, where c_r is also a leaf descendant of t. Let c_l be the paired chord of c_r. They are both incident to an upward cusp v. There are several cases to consider.

I. c_2 does not intersect c_l. In this case, c_2 can not be the S-chord of c_r by definition.

II. c_2 intersects c_l. Then c_2 is a potential S-chord of c_r. We further have two cases.

   A. c_1 intersects c_l. We know that c_1 is the winner. This implies that either the label of c_1 is greater than that of c_2, in which case c_2 can not be the S-chord of c_r (similar to the argument in Case c of the processing stage), or the origin of c_1 is contained in the component $P(c_2)$ of c_2. An equivalent way to say this is that the origin of c_1 is ahead of that of c_2 clockwise along the boundary of P starting at the left endpoint of c_l. Therefore by the same argument, c_2 is redundant.

   B. c_1 does not intersect c_l. Let c_a and c_b be the limiting chords of c_1 with their incident upward cusps v_a and v_b. We claim that there are the following two possibilities: 1. the left component of v contains the hit point of c_1; 2. there exists an upward cusp $v_a < v' < v$ whose left component contains the hip point of c_1, and whose right component contains v (Actually, the correctness of this claim does not depend on the fact that c_1 does not intersect c_l). We omit the proof of the claim in this paper.

      a. The left component of v contains the hit point of c_1. We know that c_1 does not intersect c_l. Therefore c_1 is completely contained in the left component of v. But then v would not be included by the limiting chords c_a and c_b, and therefore c would not be inserted into the LS field of t. Thus this case is not possible.

      b. There exists an upward cusp $v_a < v' < v$ whose left component contains the hip point of c_1, and whose right component contains v. Let c'_l and c'_r be the paired chords incident to v'. Clearly, c_1 can not be completely contained in the left component of v'. Therefore c_1 must intersect c'_l. We have two cases.

         i. The labels of c_1 and c_2 are the same. Since c_1 is the winner, c_1 and c_2 must be disjoint but not their components. Therefore $P(c_2)$ is contained in $P(c_2)$, implying that c_2 is redundant.

         ii. The label of c_1 is greater than that of c_2. Then any direct descendants of c_2 below c_r have their components containing that of c_1 but with no greater label. Therefore c_2 is not the necessary S-chord of c_r.

Therefore we only have to look at $O(\log n)$ chords on the path. We conclude that an insertion and extraction can be done in $O(\log n)$ time.

We devote the remaining section to the description of an $O(n \log n)$ algorithm that computes the order defined on upward cusps. First we partition P with all L- and R-chords. Each upward cusp can be uniquely associated with a region in the partition directly above it. We then define a graph G as follows. The nodes of G are all upward cusps, and an edge is added between two nodes iff their associated regions are adjacent in the partition. The graph G is a tree of size $O(n)$, and can be constructed in $O(n \log n)$ time. Following the definitions in Section 2, it is clear that an upward cusp is a top node in G iff it is a top upward cusp. We can find the set S of all top nodes of G in $O(n)$ time. We assume that all nodes of S are stored in a linked list so that they can be accessed in constant time by their indices. The output of our $O(n \log n)$ algorithm is a rooted binary tree T' such that the nodes of T' are all the upward cusps, and the *inorder* of them are exactly the order defined above. To initialize the algorithm, we choose an arbitrary top node t of G, and let it be the root of T'. Then t is deleted from G together with its two incident edges. This will split G into two components G_l and G_r such that G_l contains the left child t_l of t, and G_r contains the right child t_r of t. Let S_l and S_r be the set of top nodes of G_l and G_r respectively. Algorithmically, the two sets S_l and S_r can be obtained as follows. We choose the one of G_l and G_r, say G_l, that has fewer nodes than the other. We can accomplish this selection in $O(|G_l|)$ time by

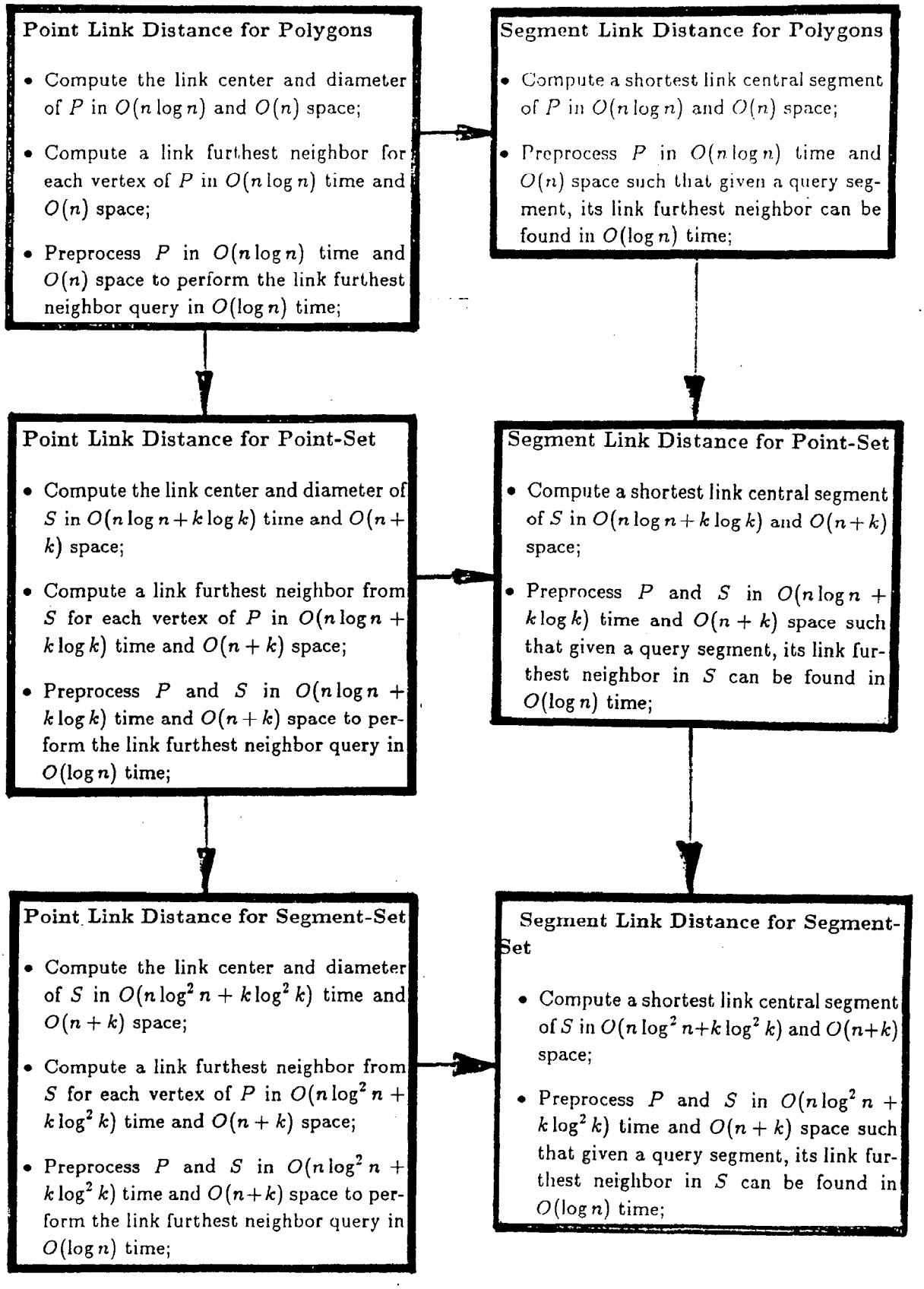

Figure 3. The table of results. The arrows indicate the directions of generalizations.

counting the nodes of G_l and G_r alternatively until one of them is exhausted. Then in additional $O(|G_l|)$ time, we can find the set S_l. Then all nodes of S_l, except possibly t_l, are deleted from S_u. The remaining nodes in S_u, possibly plus t_r, form the set S_r. After this, the algorithm can recursively choose an arbitrary node from S_l to be the left child of t in T', and a node from S_r to be the right child of t in T'_u, and so on. The time complexity of the algorithm is $O(n \log n)$.

6 Conclusion

Several extensions of the link distance problem can be solved by the described algorithm very efficiently. Let S be a set of k points (or line segments) in P. We define the link center of S to be the collection of points of P whose maximum link distance to points (or line segments) in S is the minimum, and the link diameter of S to be the maximum link distance achieved by two points (or line segments) in S. We assume that points (or line segments) in S are not obstacles for the link paths; only the boundary of P is. In general, we can redefine all the previous problems with respect to the set S (instead of the points of P). Using the sweep-chord algorithm and the techniques in [9], we can achieve the results listed in Figure 3 [7].

Acknowledgements

I am deeply indebted to Professor Joseph O'Rourke, whose suggestions considerably improved the paper. The work on this paper was partially supported by his grants from NSF, Martin-Marietta, IBM and General Motors.

References

[1] B. Chazelle and L. Guibas. Visibility and intersection problems in plane geometry. In *Proc. of First ACM Symposium on Computational Geometry*, pages 135–146, 1985.

[2] H. N. Djidjev, A. Lingas, and J.-R. Sack. *An $O(n \log n)$ Algorithm for Computing a Link Center in a Simple Polygon.* Technical Report SCS-TR-148, School of Computer Science, Carleton University, 1988.

[3] H. Edelsbrunner, L. Guibas, and J. Stolfi. Optimal point location in monotone subdivisions. *SIAM Journal on Computing*, 15(2):317–340, 1986.

[4] L. Guibas and J. Hershberger. Optimal shortest path queries in a simple polygon. In *Proc. of Third ACM Symposium on Computational Geometry*, pages 50–63, 1987.

[5] L. Guibas, J. Hershberger, D. Leven, M. Sharir, and R. Tarjan. Linear time algorithms for visibility and shortest path problems inside simple polygons. In *Proceedings of the second ACM Symposium on Computational Geometry*, pages 1–13, 1986.

[6] Y. Ke. *An Efficient Algorithm for Link Distance Problems inside a Simple Polygon.* Technical Report 28, Department of Computer Science, The Johns Hopkins University, 1987.

[7] Y. Ke. *Efficient Algorithms for Weak Visibility and Link Distance Problems in Polygons.* PhD thesis, The Johns Hopkins University, spring 1989.

[8] Y. Ke. *Testing the Weak Visibility of a Simple Polygon and Related Problems.* Technical Report 27, Department of Computer Science, The Johns Hopkins University, 1987.

[9] Y. Ke and J. O'Rourke. *Weak Visibility Problems for a Set of Points in a Simple Polygon.* Technical Report, Department of Computer Science, The Johns Hopkins University, 1987.

[10] W. Lenhart, R. Pollack, J. Sack, R. Seidel, M. Sharir, S. Suri, S. Whitesides G. Toussaint, and C. Yap. Computing the link center of a simple polygon. In *Proceedings of the third ACM Symposium on Computational Geometry*, pages 1–10, 1987.

[11] D. Johnson M. Garey, F. Preparata, and R. Tarjan. Triangulating a simple polygon. *Information Processing Letters*, 7(4):175–180, 1978.

[12] S. Suri. A linear time algorithm for minimum link paths inside a simple polygon. *Computer Vision, Graphics and Image Processing*, 35, 1986.

[13] S. Suri. *Minimum Link Paths in Polygons and Related Problems.* PhD thesis, The Johns Hopkins University, August 1987.

[14] R. Tarjan and C. Van Wyk. An $o(n \log n \log n)$ time algorithm for triangulating simple polygons. preprint (to appear in *SIAM Journal on Computing*), 1986.